UNITED STATES PATENT APPLICATION
FOR

METHOD FOR CONVERTING PIPELINE STALLS TO PIPELINE FLUSHES IN A MULTITHREADED PROCESSOR

INVENTORS:

SAILESH KOTTAPALLI
UDO WALTERSCHEIDT

PREPARED BY:

KENYON & KENYON
333 WEST SAN CARLOS STREET, SUITE 600
SAN JOSE, CALIFORNIA 95110
(408) 975-7500

Method for Converting Pipeline Stalls to Pipeline Flushes in a Multithreaded Processor

Background of the Invention

The present invention pertains to a method and apparatus for converting pipeline stalls to pipeline flushes in a processor. More particularly, the present invention pertains to flushing and tracking a stalled thread in the pipeline to allow forward progress of all non-stalling threads in a multi-threaded processor.

As is known in the art, a processor includes a variety of sub-modules, each adapted to carry out specific tasks. In one known processor, these sub-modules include the following: an instruction cache; an instruction fetch unit for fetching appropriate instructions from the instruction cache; an instruction buffer that holds the fetched instructions from the instruction cache; a scheduler that schedules the dispersal of individual instructions into separate execution pipes; execution pipes that execute the individual instructions; a pipeline control logic that monitors the activity within the execution pipes; and exception and retirement logic that checks for illegal operations and retires the exception-free instructions in program order.

Programming code to be executed by the processor can sometimes be broken down into smaller components referred to as "threads." A thread is a series of instructions whose execution achieves a given task. For example, in a video phone applications, the processor may be called upon to execute code to handle video image data as well as audio data. There may be separate code sequences whose execution is designed to handle each of these data types. Thus, a first thread may include instructions for video image data processing and a second thread may include instructions for audio data processing.

As is known in the art, simultaneous multi-threaded ("SMT") processors enable multiple threads to be active simultaneously within a single processor core. SMT implementations allow

the machine width to be optimally occupied by filling up the execution pipes with instructions from multiple threads. By definition, instructions from different threads exhibit no dependency, which enables parallel execution. During parallel execution, operation and data dependencies are handled by stalling the pipeline given the in-order nature of the processor. These stalls span from a single cycle, to address some execution latencies, to several hundreds of clock cycles in cases involving data fetches from external memories (i.e. data from the off-chip memory). Thus, thread-specific stalls with long latencies of several hundred clock cycles could result in a significant decrease in the performance of a processor.

Another shortcoming to the thread-specific stalls is the under-utilization of the multi-threaded execution pipelines. As instructions from multiple threads are interspersed amongst the execution pipes, stalling one thread results in blocking all pipes. Although the non-stalling threads can make progress, it cannot be issued to the ports of the stalled thread. As a result, the stalls have an added performance drawback for multi-threaded processors since a stalled thread effectively stalls the progress on the other non-stalling threads.

In view of the above, there is a need for an improved method and system to handle pipeline stalls in a multi-threaded processor.

Brief Description of the Drawings

Fig. 1 is a block diagram of a portion of a multi-threaded processor employing an embodiment of the present invention.

Fig. 2 is a flow diagram showing an embodiment of a method according to an embodiment of the present invention.

Detailed Description of the Drawings

Referring to Fig. 1, a block diagram of a processor system 100 (e.g. a microprocessor, a digital signal processor, or the like) operated according to an embodiment of the present invention is shown. In this embodiment, the processor is a multi-threaded processor where the execution is theoretically divided into two or more logical processors. As used herein, the term "thread" refers to an independent instruction code sequence. In this example, there are one or more execution units (i.e., multiple execution pipes 105), which may execute one or more instructions at a time. The processor system 100, however may be treated as two logical processors, a first logical processor executing instructions from the first thread (Thread 0) and a second logical processor executing instructions from the second thread (Thread 1).

In this embodiment of the processor system 100, instructions are fetched by a fetch unit 101 from memory (i.e., either from L1 cache memory 108 or from L2 memory cache 109). The instructions are then supplied to the instruction buffers, thread 0 in instruction buffer 0 (i.e., instruction buffer 102) and thread 1 in instruction buffer 1 (i.e., instruction buffer 103). The instruction buffers supply its outputs to the scheduler 104. The scheduler controls when and in what order the instructions from thread 0 or thread 1 are supplied to the execution ports 105. The pipeline control logic 106 monitors the execution of the threads and tracks the return of data from L1 memory cache 108 or L2 memory cache 109 or memory bus 110 as needed. The outputs of the execution pipes 105 are then supplied to the exception and retirement unit 107.

According to this embodiment of the present invention, a "stall-miss-flush" approach is used to flush and track a stalled thread from the execution pipes. In Fig. 1, in this example, during execution in one of the execution pipes 105, a data dependent operation or instruction in thread 0, stalls. As used herein, a data dependent operation is one where, in order to perform a

particular instruction of a thread, data needs to be loaded from memory (e.g., L1 cache 108, L2 cache 109, etc.). The pipeline control logic 106, in monitoring stalled thread 0, first attempts to retrieve the data from L1 memory cache 108, a delay of a few clock cycles. If the data is not found in the L1 memory cache 108, the pipeline control logic 106 attempts to retrieve the data from the L2 memory cache 109, which takes an additional number of clock cycles, approximately 5 to 10 cycles in this embodiment of the processor system 100. At the same time, stalled thread 0 blocks the pipeline for non-stalling thread 1 that follows in the execution pipes 105. The pipeline control logic 106 flags the instruction of the thread as a missed search (i.e., a "miss"). The stall on thread 0 is then released and the instruction is flushed through the execution pipes 105 into the exception and retirement logic 107, which allows thread 1 to continue its execution down the pipeline. Pipeline control logic 106 continues to retrieve the data from a third level of memory (e.g., random access memory coupled to memory bus 110), which may take several hundreds of clock cycles to recover. Exception and retirement logic 107 detects that thread 0 has data dependencies that have not been completed and informs fetch unit 101 to fetch the miss instruction from thread 0 once more and attempt to execute the isntructions from thread 0 again.

This time around, when the miss instruction from thread 0 reaches scheduler 104, the scheduler 104 awaits an update pipeline control logic 106 when the data needed by thread 0 has been retrieved from the memory bus. When the data is available, the scheduler starts dispersal of this instruction from thread 0 into the execution pipes 105. This time through, thread 0 can resolve its data dependency without stalling by accessing the data made available.

An example of the operation of pipeline control logic 107 in this embodiment is shown in Fig. 2. After being released by the scheduler 104, in block 201, the instruction threads are

loaded into the pipeline. In decision block 202, when a thread stalls, it is determined whether the stall is due to a dependent operation needed by the thread. If it is not (e.g., to address some execution latencies), control passes to block 204 where the thread completes execution through the pipeline. If the stall is due to a dependent data operation, control passes to decision block 203 to determine if the data can be retrieved from memory either in the L1 cache or L2 cache, rather than from external memories. If it is found in the L1 cache or L2 cache, then control passes to block 204 to await return of data and continue execution through the pipeline. If it is not found in the L1 cache or L2 cache, control shifts to block 205 where the thread is flagged as a stall-miss-flush. Control then passes to block 206 where the exception and retirement logic 107 reads the flag and re-steers the thread to the front end and eventually back to the scheduler 104. The scheduler 104, in block 207, awaits a data return indication from the pipeline control logic 106 before returning control to block 201 to load the threads into the execution pipelines. Likewise, if the thread does not stall, it continues through the pipeline for execution in a normal manner. One skilled in the art will appreciate that the amount of delay that is unacceptable to cause the thread to be flagged as a stall-miss-flush may be modified (e.g., only flag the thread as such if the dependent data is not in the L1 cache alone).

As seen from the above, a stall-miss-flush approach works to keep the utilization of the processor high by decreasing the clock cycles per instruction. This is because the non-stalling threads in the pipeline can be executed even when a long latency data dependent thread precedes it in the pipeline. During the several hundreds of clock cycles needed to resolve the data dependency, instruction from other non-stalling threads can be processed for execution. When long latency stalls are flushed, overall performance of the processing system may be enhanced.

Although a single embodiment is specifically illustrated and described herein, it will be appreciated that modifications and variations of the present invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention.